

# Intuitive and Formal Representations: The Case of Matrices

Martin Pollet<sup>1,2\*</sup>, Volker Sorge<sup>2\*\*</sup>, and Manfred Kerber<sup>2</sup>

<sup>1</sup> Fachbereich Informatik, Universität des Saarlandes, Germany  
pollet@ags.uni-sb.de, www.ags.uni-sb.de/~pollet

<sup>2</sup> School of Computer Science, The University of Birmingham, England  
{V.Sorge|M.Kerber}@cs.bham.ac.uk, www.cs.bham.ac.uk/~{vxs|mmk}

**Abstract.** A major obstacle for bridging the gap between textbook mathematics and formalising it on a computer is the problem how to adequately capture the intuition inherent in the mathematical notation when formalising mathematical concepts. While logic is an excellent tool to represent certain mathematical concepts it often fails to retain all the information implicitly given in the representation of some mathematical objects. In this paper we concern ourselves with matrices, whose representation can be particularly rich in implicit information. We analyse different types of matrices and present a mechanism that can represent them very close to their textbook style appearance and captures the information contained in this representation but that nevertheless allows for their compilation into a formal logical framework. This firstly allows for a more human-oriented interface and secondly enables efficient reasoning with matrices.

## 1 Introduction

A big challenge for formalising mathematics on computers is still to choose a representation that is on the one hand close to that in mathematical textbooks and on the other hand sufficiently formal in order to perform formal reasoning. While there has been much work on intermediate representations via a ‘mathematical vernacular’ [3, 12, 5], most of this work concentrates on representing mathematical proofs in a way that closely resembles the human reasoning style. Only little attention has been paid to an adequate representation of concrete mathematical objects, which captures all the information and intuition that comes along with their particular notation. Logicians are typically happy with the fact that such concepts can be represented in *some* way, whereas users of a formal system are more concerned with the question, how to represent a concept and how much effort is necessary to represent it. Depending on the purpose of the representation it is also important how easy it is to work with it.

In this paper we examine one particular type of mathematical objects, namely matrices. Let us first take a closer look how the matrix concept is introduced in a mathematical textbook. Lang [8, p.441] writes:

---

\* The author's work is supported by EU IHP grant Calculemus HPRN-CT-2000-00102.

\*\* The author's work is supported by a Marie-Curie Grant from the European Union.

“By an  $m \times n$  **matrix** in  $R$  one means a doubly indexed family of elements of  $R$ ,  $(a_{ij})$ , ( $i = 1, \dots, m$  and  $j = 1, \dots, n$ ), usually written in the form

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ & \cdots & \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

We call the elements  $a_{ij}$  the **coefficients** or **components** of the **matrix**.”

Mathematicians do not work with the definition alone. The definition already introduces the representation as a rectangular form in which the elements of a matrix are ordered with respect to their indices. Matrices can be viewed as collections of row or column vectors, as block matrices, and various types of ellipses are used to describe the form of a matrix. The different representations are used to make the relevant information directly accessible and ease reasoning.

Depending on the exact logical language, one would consider a matrix as a tuple consisting of a double indexed function, number of rows, number of columns, and the underlying ring. And opposed to mathematics, one has to stick to this definition during all proofs. The (logical) view that a matrix is a tuple, which mainly bears aspects of a function, is not adequate from a mathematical point of view. If we look at a concrete matrix such as a  $2 \times 2$  matrix containing only the zero element this matrix  $Z$  is a constant. This means in particular that for any matrix  $M$ , the product  $M \cdot Z$  is equal to  $Z$  without the necessity to do reasoning about tuples and lambda expression. This is analogous to the relationship between the formal logical and mathematical view of the natural number four, which logically is the ground term  $s(s(s(0)))$ , while mathematically it is the constant symbol 4.

In this paper we show how to abstract from the functional representation of concrete matrices and how to attach structural information for matrices to the formal representation by so-called annotated constants. The structural information can be used for reasoning, which simplifies proof construction since some of the reasoning steps can be expressed as computations. The connection to the formal content allows the verification of the abstract proof steps in the underlying logical calculus.

In the next section we will have a closer look at different types of matrices that we want to be able to represent. In section 3 we will introduce annotated constants as an intermediate representation for mathematical objects. In section 4 we will discuss how concrete matrices, block matrices and ellipses can be represented and manipulated as annotated constants. We conclude with a discussion of related and future work in section 5.

## 2 Matrices – Examples

In this section we give an overview of some important cases of matrices and their representations as they appear in algebra books (e.g. [8, 6]). We do not intend to give an exhaustive overview. However, we believe that the cases covered here will allow for generalisations and adaptations to others. We discuss some of the

representational issues, for which we propose solutions in the subsequent sections of the paper.

## 2.1 Concrete Matrices

A matrix is a finite set of entries arranged in a rectangular grid of rows and columns. The number of rows and columns of a matrix is often called the *size* of that matrix. The entries of a matrix are usually elements belonging to some algebraic field or ring. Matrices often occur in a very concrete form. That is, the exact number of rows and columns as well as the single entries are given. An example is the following  $2 \times 3$ -matrix:

$$M = \begin{pmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{pmatrix}$$

Matrices of this form are fairly easy to represent and handle electronically. They can simply be linearised into a list of rows, which is indeed the standard input representation of matrices for most mathematical software, such as computer algebra systems. Since both the size of the matrix is determined and all the elements are given and of a specific type, manipulations of the matrix and computations with the matrix can be efficiently performed, even if the concrete numbers are replaced by indeterminates such as  $a, b, c$ .

While concrete matrices often occur in many engineering disciplines, pure mathematics goes normally beyond concrete sizes, but will speak of matrices in a more generalised fashion.

## 2.2 Block matrices

Block matrices are matrices of fixed sizes, typically  $2 \times 2$  or  $3 \times 3$ , whose elements consist of rectangular blocks of elements of not necessarily determined size. Thus, block matrices are in effect shorthand for much larger structures, whose internal format can be captured in a particular pattern of blocks. Consider, for instance, the general matrix of size  $(n + 1) \times (n + 1)$  given as

$$M = \left( \begin{array}{c|c} a & v^T \\ \hline 0 & A \end{array} \right)$$

in which  $a \neq 0$  is a ring element (scalar),  $0$  is the zero vector of size  $n$ ,  $v^T$  is the transpose of an arbitrary vector  $v$  of size  $n$ , and  $A$  is a matrix of size  $n \times n$ .

A block matrix can be emulated with a concrete matrix by regarding its elements as matrices themselves. This can be achieved by identifying, scalars with  $1 \times 1$  matrices, vectors with  $n \times 1$  matrices, and transposed vectors with  $1 \times n$  matrices. While this enables the use of the techniques available for concrete matrices to input and represent block matrices, manipulating block matrices is not as straightforward. Since the elements do no longer belong to the same algebraic ring (or indeed to any ring), computations can only be carried out with respect to a restricted set of axioms. In particular, one has to generally forgo commutativity when computing with the single blocks. Again this can be

simulated to a certain extent. For instance, the inverse of the above matrix can be computed by using a computer algebra system that can deal with non-commutativity, as demonstrated in section 4.2. Computations concerning two block matrices, such as matrix addition or multiplication can be simulated as well. However care has to be taken that the sizes of the blocks are compatible.

### 2.3 Ellipses

While block matrices can capture simple patterns in a matrix in an abstract way, more complex patterns in generalised matrices are usually described using ellipses. Consider for instance the definition of the following matrix  $A$ :

$$A = \begin{pmatrix} a_{11} & b & \cdots & b \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & b \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

The representation stands for an infinite class of  $n \times n$  matrices such that we have a diagonal with elements  $a_{ii}$ ,  $1 \leq i \leq n$ , all elements below the diagonal are zero, while the elements above the diagonal are all  $b$ . A matrix of this form is usually called an upper triangle matrix.

In the context of matrices we can distinguish essentially two types of ellipses:

1. Ellipses denoting an arbitrary but fixed number of occurrences of the same element, such as in  $(0 \cdots 0)$ .
2. Ellipses representing a finite number of similar elements that are enumerated with respect to a given index set  $(a_1 \cdots a_n)$ <sup>1</sup>.

Both types of ellipses are primarily placeholders for a finite set of elements that are either directly given (1) or can be inferred given the index set (2).

Another important feature of ellipses in the context of matrices are their orientation. While for most mathematical objects, such as sets or vectors, ellipses can occur in exactly one possible orientation, in two-dimensional objects such as matrices we can distinguish three different orientations: horizontal, vertical, and diagonal. Thereby ellipses are not only placeholders for single rows, columns or diagonals but a combination of ellipses together can determine the composition of a whole area within the matrix. For example the interaction of the diagonal, horizontal and vertical ellipses between the three 0s in  $A$  determines that the whole area underneath the main diagonal defaults to 0.

Matrices with ellipses are well suited for manipulation and reasoning at an intuitive abstract level. However, already their mere representation poses some problems. While they can be linearised with some effort, a two-dimensional representation of the object is preferable, as this eases to determine the actual meaning of the occurring ellipses. It is even more challenging to mechanise abstract reasoning or abstract computations on matrices with ellipses.

<sup>1</sup> Here the notion of an index set is not necessarily restricted to being only a set of indices for a family of terms, but we also use it with a more general meaning of enumerating a sequence of elements as for instance in the vector containing the first  $n$  powers of  $a$ ,  $(a^1 \cdots a^n)$ .

## 2.4 Generalised Matrices

While ellipses already provide a powerful tool to express matrices in a very general form by specifying a large number of possible patterns, one sometimes wants to be even more general than that. Consider for instance the following definition of matrix  $B$ :

$$B = \begin{pmatrix} a_{11} & \star & \cdots & \star \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \star \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}. \text{ Also written as: } \begin{pmatrix} a_{11} & & \star \\ & \ddots & \\ 0 & & a_{nn} \end{pmatrix}$$

Matrix  $B$  is very similar to  $A$  above, with the one exception that the elements above the main diagonal are now arbitrary, indicated by  $\star$ , rather than  $b$ . This is a further generalisation as  $B$  now describes an upper triangle matrix of variable sizes  $n \times n$ , where we are only interested in the elements of the main diagonal  $a_{ii}$ ,  $1 \leq i \leq n$ , but we don't care about the elements above the diagonal. While such a matrix can be represented with the same representational tools as the matrix  $A$  above, it will be more difficult to deal with when we actually want to compute or reason with it.

In the following we shall describe how we can handle concrete matrices, block matrices, and ellipses. In order to extend our approach to cover generalised matrices as well, we would need to handle “don't care” symbols, in addition. We do not go into this question in this paper.

## 3 Intermediate Representation – Annotated Constants

In this section we present the concept of annotated constants, a mechanism that provides a representational layer that can both capture the properties of the intuitive mathematical representation of objects, as well as connect these objects to their corresponding representation in a formal logic framework. Annotated constants are implemented in the Omega system [10] and therefore the logical framework is Omega's simply typed lambda calculus (cf. [1]). We have first introduced annotated constants in [11] in the context of mathematical objects, such as numbers, lists, and permutations. For the sake of clarity we explain the idea in the following using the much simpler example of finite sets.

Let us assume a logical language and a ground term  $t$  in this language. Let  $c$  be a constant symbol with  $c = t$ . An annotated constant is then a triple  $(c, t, \mathbf{a})$ , in which  $\mathbf{a}$  is the annotation. The *annotation*  $\mathbf{a}$  is any object (making use of an arbitrary data structure) from which  $c$  and  $t$  can be reconstructed. Think of  $c$  as the name of the object,  $t$  as the representation within logic, and  $\mathbf{a}$  as a representation of the object outside logic.

*Finite Sets:* Finite sets have a special notation in the mathematical vernacular, for example, the set consisting of the three elements  $a$ ,  $b$ , and  $c$  is denoted by  $\{a, b, c\}$ . We can define this by giving the set a name, e.g.,  $A$ , and a definition in

logic as a ground term. Important knowledge about sets with which it is appropriate to reason efficiently is: sets are equal if they contain the same elements regardless of their order, or the union of two sets consists of the elements which are a member of one of the sets and so on. This type of set manipulation has not so much to do with logical reasoning as it has with computation. The union of two sets, for instance, can be very efficiently computed and should not be part of the process of search for a proof.

Annotated constants for finite sets are defined with the attributes

**Annotation for finite sets:** The data structure of sets of the underlying programming language is used as annotation and the elements of the set are restricted to closed terms, e.g., the set containing the three constants  $a$ ,  $b$ , and  $c$  in the concrete example.

**Constant symbol:** We give the set a name such as  $A$ . Even more appropriate for our purpose is to generate an identifier from a duplicate free ordering of the elements of the set, for the example  $A_{\{a,b,c\}}$ .

**Definition:** The definition of the set corresponds to a lambda term in higher-order logic, e.g.,  $\lambda x. (x=a \vee x=b \vee x=c)$ . In order to normalise such terms it is useful to order the elements of the set, that is, we wouldn't write the term as  $\lambda x. (x=b \vee x=a \vee x=c)$ . Since the annotation has to represent the object completely the formal definition can be constructed from the annotation.

The basic functionality for handling annotated constants is implemented on the term level of the Omega system. In first approximation, an annotated constant is a constant with a definition and has the type of its defining term  $t$ . As such it could be replaced by its defining term during the proof or when expanding the proof to check formal correctness. Typically, this is not done, but annotated constants are manipulated via their annotations. The defining term of an annotated term is used only when necessary.

The manipulation of operations and verification of properties is realised as procedural annotations to functions and predicates. A procedural annotation is a triple  $(f, \mathbf{p}, T)$ , where  $f$  is a function or predicate of the logical language,  $\mathbf{p}$  is a procedure of the underlying programming language with the same number of arguments as  $f$  and  $T$  is a specification (or tactic) for the construction of a formal proof for the manipulation performed by  $\mathbf{p}$ . The procedure  $\mathbf{p}$  checks its arguments, performs the simplification, and returns a simplified constant or term together with possible conditions for this operation.

For example, the procedure for the union of concrete sets  $\{a, b\} \cup \{c, d\}$  checks whether the arguments are annotated constants for concrete sets, and returns the annotated constant which has the concatenation of  $\{a, b\}$  and  $\{c, d\}$  as annotation. Analogously the property  $\{1, 2, 3\} \subset \mathbb{Z}$  holds, when all elements of the annotation of the set are constants which have as annotation an integer as data structure.

The proof specification  $T$  is used to formally justify the performed step. Thereby an annotated constant is expanded to its formal definition and the computation is reconstructed by tactic and theorem applications. This expansion will be done only when a low level formal proof is required, certainly not during proof search.

### What are the advantages of using annotated constants?

Firstly, annotated constants provide an intermediate representation layer between the intuitive mathematical vernacular and a formal system. With annotated constants it is possible to abstract from the formal introduction of objects, allow the identification of certain classes of objects and enable the access of relevant knowledge about an object directly. Annotations can be translated into full formal logic expressions when necessary, but make it possible to work and reason with mathematical objects in a style that abstracts from the formal construction.

Secondly, annotations allow for user friendly input and output facilities. We extended Omega's input language to provide a markup for an annotated constant to indicate the type of the object it represents. For each kind of annotated constant the term parser is extended by an additional function, which parses annotations and transforms these annotations into an internal representation. During parsing additional properties can be checked and errors in the specification can be detected. In this way it is possible to extend syntactic type checking. An additional output function for each kind of annotated constant allows to have different display forms for presenting formulas to the user.

Thirdly, procedural annotations enable an efficient manipulation of annotated constants. These procedures can access information without further analysis on (lambda) terms (which define annotated constants formally) and allows to compute standard functions and relations very efficiently. These operations and properties become a computation on the data structures of annotated constants.

## 4 Matrices as Annotated Constants

In this section we show how annotated constants can be used to implement the different representations for matrices presented in section 2.

### 4.1 Concrete Matrices

Concrete matrices of fixed sizes, for example,

$$M = \begin{pmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{pmatrix}$$

can be represented in higher-order logic as a 4-tuple:  $(f, 2, 4, \mathbb{Z})$  where  $f$  is the lambda expression<sup>2</sup>

```
λi λj. if    i = 1 ∧ j = 1 then 3
      elseif i = 1 ∧ j = 2 then 2
      elseif i = 1 ∧ j = 3 then 7
      elseif i = 2 ∧ j = 1 then 1
      elseif i = 2 ∧ j = 2 then 0
      else 4.
```

---

<sup>2</sup> The conditional `if P then m else n` can be defined in higher-order logic by the description operator  $\iota$ . The expression  $\iota y. S(y)$  denotes the unique element  $c$  such that  $S(c)$  holds, if such a unique element exists. A conditional can thus be defined by  $\iota k. (P \wedge (k = m)) \vee (\neg P \wedge (k = n))$ , which returns  $m$  if  $P$  holds, else  $n$  (for more details see [1]).

When we look at the concrete matrix the information connected to this representation is, that the position of all the elements is specified and that the number of rows and columns of the matrix are immediately perceivable. When we look at the formal representation, then even to access an element at a certain position requires reasoning, even non-trivial reasoning when, for example, the first condition is given as  $f \lambda i \lambda j \bullet$  if  $P(i, j)$  then 3 elseif ... where  $P(i, j)$  is some equivalent formulation of  $i = 1 \wedge j = 1$ .

Also in order to multiply two concrete matrices, reasoning about the corresponding lambda expressions is necessary. For instance, the transpose of  $M$  is

$$M^T = \begin{pmatrix} 3 & 1 \\ 2 & 0 \\ 7 & 4 \end{pmatrix}$$

which can be represented as a 4-tuple  $(f^T, 4, 2, \mathbb{Z})$  where  $f^T$  is the function you get by swapping the arguments of  $f$ , i.e.  $\lambda j \lambda i$  rather than  $\lambda i \lambda j$ . The product  $M \otimes M^T$  is a matrix  $(f *_4 f^T, 2, 2, \mathbb{Z})$ , in which the function is computed component wise as the sum of products of ring elements, that is,  $\lambda i \lambda j \bullet$  if  $i = 1 \wedge j = 1$  then  $3 \cdot 3 + 2 \cdot 2 + 7 \cdot 7$  elseif ..., which requires considerable reasoning to arrive at the result. We argue that although this can be done in logic, it is not appropriate, analogously as it is not appropriate to compute a product such as  $20 \cdot 15$  by reasoning with the definition of  $\cdot$  in the constructors 0 and  $s$  over the natural numbers.

We therefore define annotated constants for concrete matrices as follows:

**Annotation for concrete matrices:** The data structure of arrays, where the elements are in the logical language and all of them have the same type  $\alpha$ . All places of the array must be filled with constants of the logical language, as for instance in our example matrix  $M$ .

**Constant:** A constant  $A$  of the logical language of type  $\mathbb{Z} \times \mathbb{Z} \rightarrow \alpha$ .

**Definition:** The lambda expression representing a *double indexed function* of the form  $\lambda i \lambda j \bullet f(i, j)$ , where  $i$  and  $j$  range over the integer intervals  $[1, m]$  and  $[1, n]$ , respectively and every  $f(i, j)$  is an element of a ring  $F$ . In other words  $f(i, j)$  corresponds to a matrix entry in the  $i^{\text{th}}$  column and the  $j^{\text{th}}$  row. To guarantee that a double indexed function actually constitutes a matrix it has to fulfil the property  $Matrix(f, m, n, F) \equiv \forall i \in [1, m], j \in [1, n] : f(i, j) \in F$ . A concrete example for such a lambda expression is the double indexed function representing  $M$  above.

For annotated constants representing concrete matrices the operations for summation and multiplication of matrices, scalar multiplication, and transposing a matrix are annotated by corresponding procedures. With the tactic *simplify* which applies all possible simplifications specified in annotated procedures, a proof step performing the matrix multiplication  $M \otimes M^T$  is:

$$\begin{array}{ll}
L_1. & \{0, 1, 2, 3, 4, 7\} \in R \quad (\text{open}) \\
L_2. & \text{Ring}(R, +, \cdot) \quad (\text{open}) \\
L_3. & P = \begin{pmatrix} 62 & 31 \\ 31 & 17 \end{pmatrix} \quad (\text{open}) \\
L_4. & P = \begin{pmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{pmatrix} \otimes \begin{pmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{pmatrix}^T \quad (\text{simplify } L_1, L_2, L_3)
\end{array}$$

The line  $L_3$  contains the matrix which is the result of the simplification. Since the matrices consist of integers, which are annotated constants again, the simplification can compute the result of arithmetic operations on integers. The lines  $L_1$  and  $L_2$  contain the side conditions of the computation. Note that the necessary conditions on the size of the matrices  $\text{Matrix}(\left(\begin{smallmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{smallmatrix}\right), 2, 3, R)$  and  $\text{Matrix}(\left(\begin{smallmatrix} 3 & 2 & 7 \\ 1 & 0 & 4 \end{smallmatrix}\right)^T, 3, 2, R)$  are available from the annotation and thus can be checked during the expansion of the tactic *simplify*.

## 4.2 Block matrices

A block matrix of the form  $M = \left(\begin{array}{c|c} a & v^T \\ \hline 0 & A \end{array}\right)$  can be formally expressed in logic as a tuple  $(f, n + 1, n + 1, F)$ , in which  $f$  is a function from the indices into the ring,  $\{1, \dots, n + 1\} \times \{1, \dots, n + 1\} \rightarrow F$  defined as

$$\begin{array}{ll}
\lambda i \lambda j. & \text{if } i = 1 \wedge j = 1 \quad \text{then } a \\
& \text{elseif } i = 1 \wedge 2 \leq j \leq n + 1 \quad \text{then } v_{j-1} \\
& \text{elseif } 2 \leq i \leq n + 1 \wedge j = 1 \quad \text{then } 0 \\
& \text{else} \quad \quad \quad a_{i-1, j-1}
\end{array}$$

If we assume  $a \neq 0$  and  $\det(A) \neq 0$ , we can then show that the set of matrices of the above form constitute a subgroup of the group of invertible matrices with respect to matrix multiplication. This is, however, not straightforward using the lambda expression, since a lot of the structural information is needed for the argument, which is lost in the lambda term. What we really want to do is to lift the argument about  $2 \times 2$  block matrices to a sound argument about general matrices.

In effect the argument can be collapsed into a single computation. By emulating the block matrix with a  $2 \times 2$  matrix over a non-commutative ring we can compute its inverse using the computer algebra system Mathematica [13] together with the NCAAlgebra package [9] for non-commutative algebra. If we replace the block matrix with a matrix of the form

$$\begin{pmatrix} a & b \\ 0 & c \end{pmatrix}, \text{ the corresponding inverse matrix is } \begin{pmatrix} a^{-1} & -a^{-1} \cdot b \cdot c^{-1} \\ 0 & c^{-1} \end{pmatrix}.$$

Note that  $-a^{-1} \cdot b \cdot c^{-1}$  can not be further simplified to  $-\frac{b}{a \cdot c}$ , since matrix multiplication is non-commutative. This computation on concrete matrices can be used to determine the inverse of the original block matrix by simply substituting  $v^T$  for  $b$  and  $A$  for  $c$ :

$$\begin{pmatrix} a^{-1} & -a^{-1} \cdot v^T \cdot A^{-1} \\ 0 & A^{-1} \end{pmatrix}$$

With the additional fact that  $a^{-1}$  and  $A^{-1}$  exist if and only if  $a \neq 0$  and  $\det(A) \neq 0$ , the property can be proved.

Block matrices are implemented as annotated constants in the following way:

**Annotation for block matrices:** The data structure of  $2 \times 2$  arrays  $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$

where the elements  $A, B, C, D$  are either annotated constants for matrices or double indexed lambda functions. All elements must have the same type. In addition, for annotated constants representing matrices the following conditions must hold for the number of rows  $row(A) = row(B) = r_1$ ,  $row(C) = row(D) = r_2$ , and  $col(A) = col(C) = c_1$ ,  $col(B) = col(D) = c_2$  for the number of columns.

**Constant:** A constant of type  $\mathbb{Z} \times \mathbb{Z} \rightarrow \alpha$  representing the matrix.

**Definition:** Block matrices are expanded into a lambda term of the form

$$\lambda i \lambda j. \text{ if } \begin{array}{ll} 1 \leq i \leq r_1 & \wedge 1 \leq j \leq c_1 \text{ then } A(i, j) \\ \text{elseif } 1 \leq i \leq r_1 & \wedge c_1 + 1 \leq j \text{ then } B(i, j - c_1) \\ \text{elseif } r_1 + 1 \leq i & \wedge 1 \leq j \leq c_1 \text{ then } C(i - r_1, j) \\ \text{else [i.e., } r_1 + 1 \leq i & \wedge c_1 + 1 \leq j] \quad D(i - r_1, j - c_1), \end{array}$$

where the  $A(\cdot), B(\cdot), C(\cdot), D(\cdot)$  denote double indexed functions, possibly generated by expansion of concrete matrices.

The annotated constants for block matrices allow us to combine matrices given by lambda expressions with concrete matrices. The operations for matrices then work directly on the individual blocks of the block matrices. Given double indexed functions  $u_{ij}, v_{ij}, A_{ij}, B_{ij}$  the tactic *simplify* applied to the formula in  $L_8$  results in the following proof situation:

$$\begin{array}{ll} L_1. & \text{Matrix}(u_{ij}, 1, 2, R) \quad (\text{open}) \\ L_2. & \text{Matrix}(v_{ij}, 1, 2, R) \quad (\text{open}) \\ L_3. & \text{Matrix}(B_{ij}, 2, 2, R) \quad (\text{open}) \\ L_4. & \text{Matrix}(A_{ij}, 2, 2, R) \quad (\text{open}) \\ L_5. & \{0, 1\} \in R \quad (\text{open}) \\ L_6. & \text{Ring}(R, +, \cdot) \quad (\text{open}) \\ L_7. & M = \left( \begin{array}{c|c} (1) & u_{ij} \oplus (v_{ij} \otimes B_{ij}) \\ \hline \begin{pmatrix} 0 \\ 0 \end{pmatrix} & A_{ij} \otimes B_{ij} \end{array} \right) \quad (\text{open}) \\ L_8. & M = \left( \begin{array}{c|c} (1) & v_{ij} \\ \hline \begin{pmatrix} 0 \\ 0 \end{pmatrix} & A_{ij} \end{array} \right) \otimes \left( \begin{array}{c|c} (1) & u_{ij} \\ \hline \begin{pmatrix} 0 \\ 0 \end{pmatrix} & B_{ij} \end{array} \right) \quad (\text{simplify } L_1, \dots, L_7) \end{array}$$

Line  $L_7$  contains the result of the matrix multiplication and lines  $L_1$  to  $L_6$  the side conditions on the objects involved, which cannot be inferred from the annotations.

We describe the simplification stepwise. First the sub-blocks of the matrices are multiplied, resulting in

$$\left( \begin{array}{c|c} ((1) \otimes (1)) \oplus \left( v_{ij} \otimes \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) & ((1) \otimes u_{ij}) \oplus (v_{ij} \otimes B_{ij}) \\ \hline \left( \begin{pmatrix} 0 \\ 0 \end{pmatrix} \otimes (1) \right) \oplus \left( A_{ij} \otimes \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) & \left( \begin{pmatrix} 0 \\ 0 \end{pmatrix} \otimes u_{ij} \right) \oplus (A_{ij} \otimes B_{ij}) \end{array} \right).$$

Already at this point the side conditions regarding the double indexed functions  $u_{ij}, v_{ij}, A_{ij}, B_{ij}$  are generated. The requirements for the size can be reconstructed from the sizes of the concrete matrices together with the condition from the matrix multiplication. Then simplification is applied to the content of each sub-block, starting with simplification of the arguments of an operation.

For concrete matrices operations are performed as described in section 4.1. For the simplification of operations containing both concrete matrices and double indexed functions we only consider the following cases:

- multiplications involving the zero matrix (i.e., a concrete matrix containing only the zero element of the underlying ring) are replaced by the zero matrix;
- summations with the zero matrix are replaced by the double indexed function;
- multiplication with a concrete diagonal matrix, containing the same element on the diagonal is replaced by scalar multiplication with said diagonal element.

Simplifying the above block matrix with respect to the rules for multiplication then yields

$$\left( \frac{(1) \oplus (0)}{\begin{pmatrix} 0 \\ 0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \end{pmatrix}} \middle| \frac{1 \cdot u_{ij} \oplus (v_{ij} \otimes B_{ij})}{\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \oplus (A_{ij} \otimes B_{ij})} \right)$$

Further simplification employs the rules for addition and also scalar multiplication on  $1 \cdot u_{ij}$  resulting in the formula given in  $L_7$  of the above proof.

The simplification for operations on matrices with mixed representations could also be carried out differently, namely by introducing concrete matrices, that is

$$\begin{pmatrix} a \\ b \end{pmatrix} \otimes u_{ij} \rightarrow \begin{pmatrix} a \cdot u_{11} & a \cdot u_{12} \\ b \cdot u_{11} & b \cdot u_{12} \end{pmatrix}$$

and then apply simplification on the elements of the matrix. For  $a = b = 0$  the result will be the same as for our simplification, but in the general case, the result is a concrete matrix having elements of the double indexed mixed with the elements of the concrete matrix. This means the structure of the initial blocks would be lost or hard to recognise.

While our example works with  $3 \times 3$  matrices represented as  $2 \times 2$  block matrices, the argument can be extended to arbitrary  $n \times n$  matrices still represented as  $2 \times 2$  block matrices of the form:

$$\left( \frac{(1)}{\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}} \middle| \frac{u_{ij}}{A_{ij}} \right)$$

But in order to do this we need an adequate treatment of ellipses.

### 4.3 Ellipses

While block matrices already allow us to combine concrete matrices using arbitrary double indexed functions, they only enable us to combine rectangular shapes. Using ellipses we can further generalise the representation of matrices. We then need to generalise also the simplifications introduced in the last section to matrices with fixed but arbitrary sizes. If we consider our example matrix

$$A = \begin{pmatrix} a_{11} & b & \cdots & b \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & b \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

then  $A$  can be represented in higher-order logic as a 4-tuple:  $(f, n, n, \mathbb{Z})$  where  $f$  corresponds to the lambda expression:

$$\lambda i \lambda j. \begin{cases} \text{if } i = j \text{ then } a_{ij} \\ \text{if } i < j \text{ then } b \\ \text{else} & 0 \end{cases}$$

Compared to the concrete instances above, the higher-order representation is concise. Nevertheless, in mathematics one develops particular methods for reasoning with matrices of non-concrete sizes, which follow a particular pattern, such as triangle matrices, diagonal matrices, and step matrices. Since these patterns are not necessarily obvious given the lambda term alone it is desirable to have the explicit representation of matrices with ellipses available for reasoning.

Ellipses are realised using annotated constants as well. They are categorised into horizontal, vertical, and diagonal ellipses and have the following four attributes that connect them within the matrix and determine their meaning:

**Begin:** A concrete element that marks the start of the ellipsis.

**End:** A concrete element that marks the end of the ellipsis.

**Element:** The element the ellipsis represents; this can either be a concrete element such as 0 or  $b$ , or a schematic element such as  $a_{\chi, \xi}$ . Here  $\chi$  and  $\xi$  are schematic variables that indicate that they are iterated over.

**Range:** In case the element is concrete (e.g. 0 or  $b$ ), no range is given. If the ellipsis has a schematic term as element, the integer ranges for the necessary schematic variables are given. In our example we have  $1 \leq \xi \leq n$  and  $1 \leq \chi \leq n$  meaning that both  $\xi$  and  $\chi$  take values from 1 to  $n$  with increment 1.

The values for the attributes are determined during parsing of the expression. Thereby not all combinations of ellipses are permitted. Essentially, we distinguish three basic modules a matrix can consist of:

1. points, i.e. single concrete elements.
2. lines, i.e. an ellipsis or a sequence of ellipses of the same form together with concrete elements as start and end points. An example of a line comprised of more than one ellipsis is for instance the main diagonal of  $A$  where two diagonal ellipses constitute a line from  $a_{11}$  to  $a_{nn}$ .

3. triangles, i.e. a combination of a horizontal, a vertical and a diagonal line. Since we only allow for one type of diagonal ellipsis, the we can get exactly

two different types of isosceles right triangles:  $\begin{matrix} \bullet & \cdots & \bullet \\ & \ddots & \vdots \\ & & \bullet \end{matrix}$   $\begin{matrix} & & \bullet \\ & \ddots & \vdots \\ \bullet & \cdots & \bullet \end{matrix}$

Both start and end elements of an ellipsis are determined by searching for a concrete element in the respective direction (i.e., left/right, up/down, etc.) while ignoring other ellipses. Both element and range are computed given the start and end: If the start and end terms are the same then this term is taken to be the element the ellipsis represents and no range needs to be computed. In case they are not the same we try to compute a schematic term using unification. Although the unification fails it will provide us with a disagreement set on the two terms, which can be used to determine the position of possible schematic variables. If the disagreement set is sensible, that is, it consists only of terms representing integers, the schematic term is constructed and the ranges for the schematic variables are computed.

We illustrate how exactly ranges are computed with the help of some examples. Consider the vector  $(a_1^n \cdots a_n^1)$ , the schematic term is then  $a_\xi^\chi$  and the ranges are  $\xi \in \{1, \dots, n\}$  and  $\chi \in \{n, \dots, 1\}$ . Since these ranges are both over the integer and compatible, in the sense that they are of the same length, the ellipsis is fully determined. As an example of incompatible ranges consider the vector  $(a_1^k \cdots a_n^1)$ ; without further information on  $n$  and  $k$  the computation of the ellipsis will fail. Currently the increment of the range sets is always assumed to be 1. The computation of possible index sets is currently more a pragmatic one and rather simple. It is definitely not complete since there are many more possible uses of indices conceivable in our context.

An ellipsis is said to be *computable* if we can determine both begin and end element, if the element is either a concrete element or a schematic term, and if sensible and compatible integer ranges can be computed. Otherwise parsing of an ellipsis will fail. An ellipsis within a matrix gets the same type as the elements of that matrix. This means ellipses are generally treated as ordinary terms of the matrix, in particular with respect to input, display, and internal representation. For instance, our example matrix  $A$  is input as the  $4 \times 4$  matrix

$$\begin{pmatrix} (a(1,1) & b & \text{hdots} & b & ) \\ (0 & \text{ddots} & \text{ddots} & \text{vdots} & ) \\ (\text{vdots} & \text{ddots} & \text{ddots} & b & ) \\ (0 & \text{hdots} & 0 & a(n,n)) \end{pmatrix}$$

and is also represented internally as a  $4 \times 4$  array. However, the simplifications can use the information provided by the ellipsis during the reasoning process.

When a matrix containing ellipses is expanded into a lambda term the expansion algorithm translates the ellipses into appropriate conditions for the if-then-else statements. Thereby the matrix is first scanned and broken down into its components, i.e. points, lines, and triangles. These can then be associated with corresponding index sets and translated into a lambda expression. For instance the diagonal ellipsis in our example matrix  $A$  above can be simply translated into the conditional if  $i = j$  then  $a_{ij}$ , while the areas above and below the main

diagonal where a horizontal, a vertical, and a diagonal ellipsis bound the area in which all the elements are either 0 or  $b$ . In an additional optimisation step neighbouring triangles are compared and can be combined to form rectangular areas.

The simplification for operations on matrices are extended to the cases where matrices contain ellipses. For example, the sum of matrices where both matrices contain ellipses at the same positions results in a matrix containing the sum of concrete elements and the ellipses between those elements. The multiplication of a diagonal matrix containing the same element on the diagonal is reduced to scalar multiplication with this element.

## 5 Conclusions

Formal representations of mathematical objects often do not model all important aspects of that object. Especially some of the structural properties may be lost or hard to recognise and reconstruct. In our work we investigated these structural properties for the case of matrices where there exist different representations for different purposes. Each representation has certain reasoning techniques attributed to it.

We modelled the structural knowledge about concepts with the help of annotations, which are used to identify objects and to store information about them. We implemented the different representations for matrices as annotated constants and showed how basic simplifications are performed. The representations for block matrices and ellipses allow us to represent matrices of a general form. Annotations are also used for manipulations of objects. Instead of deduction on formulas, many manipulations can be reduced to computations on annotations. Since we are able to express general matrices, we can express general properties and theorems based on our formalism. With simplifications performed on generalised matrices we are now able to express complex reasoning in the form of computational steps. In future work we want to investigate how this can further aid in the construction of actual proofs. Remember that we currently deal annotated constants, that is, only with ground terms. Thus, it would be useful to extend the work in a way that allows also to deal with variables.

Annotations preserve the correctness by their implementation as constants of the formal language. The proof construction is split into a phase where steps are performed based on the richer knowledge contained in annotations and a verification phase where these steps are expanded to calculus level proofs. The expansion is currently only implemented for a subset of the operations. The expansion mechanism for proof steps using annotations needs to be simplified and generalised. The use of canonical forms should help keeping this expansion simple.

Our work compares to de Bruijn's idea of a mathematical vernacular [3], which should allow to write everything mathematicians do in informal reasoning, in a computer assisted system as well. In this tradition, Elbers looked in [4] at aspects of connecting informal and formal reasoning, in particular the integration of computations into formal proofs. Kamareddine and Nederpelt [5]

have formalised de Bruijn's idea further. While the approach to a mathematical vernacular is general, to our knowledge no attempt has been made to incorporate concrete objects like matrices directly. In the Theorema system Kutsia [7] has worked with sequence variables which stand for symbols of flexible arity. Sequence variables have some similarities to our ellipses. However, as opposed to sequence variables our ellipses allow only fixed interpretations. Moreover sequence variables can be viewed as an extension of the logical system which allows to deal with these expressions within the logic. The main emphasis of our work is to allow for representation within logic and extra-logical manipulation of expressions at the same time. Bundy and Richardson [2] introduced a general treatment for reasoning about lists with ellipses in a way that they consider an ellipsis as a schema which stands for infinitely many expressions and a proof about ellipses stands for infinitely many proofs, which can be generated from a meta-argument.

## References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer, 2nd edition, 2002.
2. A. Bundy and J. Richardson. Proofs about lists using ellipsis. In *Proc. of the 6th LPAR*, volume 1705 of *LNAI*, p. 1–12. Springer, 1999.
3. N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In *Selected Papers on Automath*, p. 865–935. Elsevier, 1994.
4. H. Elbers. *Connecting Informal and Formal Mathematics*. PhD thesis, Eindhoven University of Technology, 1998.
5. F. Kamareddine and R. Nederpelt. A refinement of de Bruijn's formal language of mathematics. *Journal of Logic, Language and Information*, 13(3):287–340, 2004.
6. M. Köcher. *Lineare Algebra und analytische Geometrie*. Springer, 1992.
7. T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Proc. of AICS'2002 & Calculemus'2002*, volume 2385 of *LNAI*. Springer, 2002.
8. S. Lang. *Algebra*. Addison-Wesley, Second Edition, 1984.
9. NCAlgebra 3.7 — A Noncommutative Algebra Package for Mathematica. Available at <http://math.ucsd.edu/~ncalg/>.
10. Omega Group. Proof development with Omega. In *Proc. of CADE-18*, volume 2392 of *LNAI*, p. 143–148. Springer, 2002.
11. M. Pollet and V. Sorge. Integrating computational properties at the term level. In *Proc. of Calculemus'2002*, p. 78–83, 2003.
12. M. Wenzel and F. Wiedijk. A Comparison of Mizar and Isar. *J. of Automated Reasoning*, 29(3–4):389–411, 2002.
13. S. Wolfram. *The Mathematica book*. Wolfram Media, Inc., 5th edition, 2003.